

# Nios<sup>®</sup> II

## Nios II Custom Instruction User Guide

---



101 Innovation Drive  
San Jose, CA 95134  
(408) 544-7000  
<http://www.altera.com>

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



Printed on recycled paper

UG-N2CSTNST-1.4



I.S. EN ISO 9001

---

<b>About this User Guide</b> .....	<b>3</b>
Revision History .....	3
How to Contact Altera .....	3
Typographic Conventions .....	2-4
<b>Chapter 1. Nios II Custom Instruction Overview</b>	
Introduction .....	1-1
Custom Instruction Overview .....	1-2
Implementing Custom Instruction Hardware .....	1-2
Implementing Custom Instruction Software .....	1-3
Custom Instruction Architectural Types .....	1-4
Combinatorial Custom Instruction Architecture .....	1-5
Combinatorial Port Operation .....	1-6
Multi-Cycle Custom Instruction Architecture .....	1-7
Multi-Cycle Port Operation .....	1-8
Extended Custom Instruction Architecture .....	1-9
Extended Custom Instruction Port Operation .....	1-10
Internal Register File Custom Instruction Architecture .....	1-11
Internal Register File Custom Instruction Port Operation .....	1-12
External Interface Custom Instruction Architecture .....	1-12
<b>Chapter 2. Software Interface</b>	
Introduction .....	2-1
Custom Instruction Examples .....	2-1
Built-In Functions and User-Defined Macros .....	2-2
Custom Instruction Assembly Software Interface .....	2-4
<b>Chapter 3. Implementing a Nios II Custom Instruction</b>	
Introduction .....	3-1
Implementing Custom Instruction Hardware in SOPC Builder .....	3-1
Open the Component Editor .....	3-1
Add the Top-Level HDL File .....	3-3
Import Simulation Files .....	3-4
Configure the Custom Instruction Ports .....	3-4
Set Up Additional Interfaces .....	3-5
Set the Component Group Name .....	3-6
Save the Custom Instruction .....	3-7
Generate the SOPC Builder System and Compile in Quartus II Software .....	3-9
Accessing the Custom Instruction from Software .....	3-9

## **Appendix A. Custom Instruction Templates**

Overview .....	A-1
VHDL Template .....	A-1
Verilog HDL Template .....	A-2

## **Appendix B. Custom Instruction Built-In Functions**

Overview .....	B-1
Built-In Functions Returning void .....	B-1
Built-in Functions Returning int .....	B-1
Built-in Functions Returning float .....	B-2
Built-in Functions Returning a Pointer .....	B-2

## **Appendix C. Porting First- Generation Nios Custom Instructions to Nios II Systems**

Overview .....	C-1
Hardware Porting Considerations .....	C-1
Software Porting Considerations .....	C-1



# About this User Guide

**Revision History** The table below displays the revision history for chapters in this User Guide.

Nios II Custom Instruction User Guide Revision History			
Chapter	Date	Version	Changes Made
2	May 2007	1.4	Add title and core version number to page footers
All	May 2007	1.3	Minor corrections to terminology and usage.
1	May 2007	1.3	Describe new component editor import flow.
3	May 2007	1.3	Remove tutorial design
All	December 2004	1.2	Updates for the Nios® II version 1.1 release.
All	September 2004	1.1	Updates for the Nios II version 1.01 release.
All	May 2004	1.0	First release of custom instruction user guide for the Nios II processor.

## How to Contact Altera

For the most up-to-date information about Altera® products, refer to the following table.

Information Type	Contact (1)
Technical support	<a href="http://www.altera.com/mysupport/">www.altera.com/mysupport/</a>
Technical training	<a href="http://www.altera.com/training/custrain@altera.com">www.altera.com/training/custrain@altera.com</a>
Product literature	<a href="http://www.altera.com/literature/">www.altera.com/literature/</a>
Altera literature services	<a href="mailto:literature@altera.com">literature@altera.com</a>
FTP site	<a href="http://ftp.altera.com">ftp.altera.com</a>

*Note to table:*

(1) You can also contact your local Altera sales office or sales representative.

## Typographic Conventions

This document uses the typographic conventions shown below.

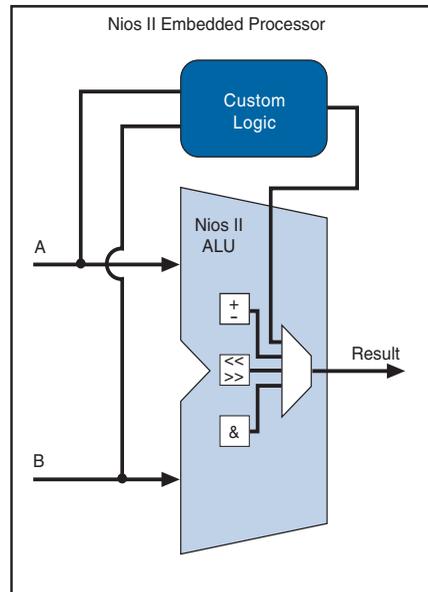
Visual Cue	Meaning
<b>Bold Type with Initial Capital Letters</b>	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <b>Save As</b> dialog box.
<b>bold type</b>	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: <b>f<sub>MAX</sub></b> , <b>qdesigns</b> directory, <b>d:</b> drive, <b>chiptrip.gdf</b> file.
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75: High-Speed Board Design</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t<sub>PIA</sub></i> , <i>n + 1</i> .  Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <file name>, <project name>.pdf file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
“Subheading Title”	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: “Typographic Conventions.”
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn.  Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\qdesigns\tutorial\chiptrip.gdf. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword SUBDESIGN), as well as logic function names (e.g., TRI) are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets are used in a list of items when the sequence of the items is not important.
	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
	A warning calls attention to a condition or possible situation that can cause injury to the user.
	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

## Introduction

With the Altera Nios II embedded processor, you as the system designer can accelerate time-critical software algorithms by adding custom instructions to the Nios II instruction set. With custom instructions, you can reduce a complex sequence of standard instructions to a single instruction implemented in hardware. You can use this feature for a variety of applications, for example, to optimize software inner loops for digital signal processing (DSP), packet header processing, and computation-intensive applications. The Nios II configuration wizard, part of the Quartus® II software's SOPC Builder, provides a graphical user interface (GUI) used to add up to 256 custom instructions to the Nios II processor.

The custom instruction logic connects directly to the Nios II arithmetic logic unit (ALU) as shown in [Figure 1-1](#).

**Figure 1-1. Custom Instruction Logic Connects to the Nios II ALU**



This chapter includes the following information:

- Description of the Nios II custom instruction feature
- Requirements for implementing a custom instruction in hardware and software
- Definition of custom instruction architectural types

For information regarding the custom instruction software interface, refer to [Chapter 2, Software Interface](#). For step-by-step instructions for implementing a custom instruction, see [Chapter 3, Implementing a Nios II Custom Instruction](#).

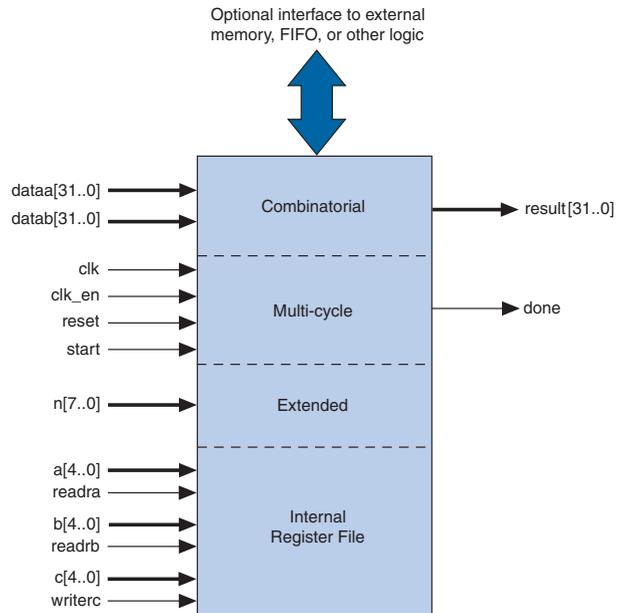
## Custom Instruction Overview

With Nios II custom instructions, you can take full advantage of the flexibility of FPGAs to meet system performance requirements. Custom instructions allow you to add custom functionality to the Nios II processor ALU.

Nios II custom instructions are custom logic blocks adjacent to the ALU in the processor's data path. Custom instructions give you the ability to tailor the Nios II processor core to meet the needs of a particular application. You have the ability to accelerate time critical software algorithms by converting them to custom hardware logic blocks. Because it is easy to alter the design of the FPGA-based Nios II processor, custom instructions provide an easy way to experiment with hardware/software tradeoffs at any point in the design process.

### Implementing Custom Instruction Hardware

[Figure 1-2](#) is a hardware block diagram of a Nios II custom instruction.

**Figure 1–2. Hardware Block Diagram of a Nios II Custom Instruction**

The basic operation of Nios II custom instruction logic is to receive input on the `dataa` and/or `datab` port, and drive out the result on its `result` port. You generate the custom instruction logic that produces the results.

The Nios II processor supports different architectural types of custom instructions. [Figure 1–2](#) lists the additional ports that accommodate different architectural types. Only the ports used for the specific custom instruction implementation are required.

[Figure 1–2](#) also shows an optional interface to external logic. The interface to external logic allows you to include a custom interface to system resources outside of the Nios II processor data path.

## Implementing Custom Instruction Software

The Nios II custom instruction software interface is simple and abstracts the details of the custom instruction from the programmer. For each custom instruction, the Nios II integrated development environment (IDE) generates a macro in the system header file, `system.h`. You can call the macro from C or C++ application code as a normal function call and

you do not need to program assembly to access custom instructions. Software can also invoke custom instructions in Nios II processor assembly language.



For more information, refer to [Chapter 2, Software Interface](#).

## Custom Instruction Architectural Types

There are different custom instruction architectures available to suit the application’s requirements. The architectures range from a simple, single-cycle combinatorial architecture to an extended variable-length, multi-cycle custom instruction architecture. The chosen architecture determines what the hardware interface looks like.

[Table 1–1](#) shows custom instruction architectural types, application, and the associated hardware ports.

<b>Architectural Type</b>	<b>Application</b>	<b>Hardware Ports</b>
Combinatorial	Single clock cycle custom logic blocks	<ul style="list-style-type: none"> <li>• dataa [31..0]</li> <li>• datab [31..0]</li> <li>• result [31..0]</li> </ul>
Multi-cycle	Multi clock cycle custom logic block of fixed or variable durations	<ul style="list-style-type: none"> <li>• dataa [31..0]</li> <li>• datab [31..0]</li> <li>• result [31..0]</li> <li>• clk</li> <li>• clk_en</li> <li>• start</li> <li>• reset</li> <li>• done</li> </ul>
Extended	Custom logic blocks that are capable of performing multiple operations	<ul style="list-style-type: none"> <li>• dataa [31..0]</li> <li>• datab [31..0]</li> <li>• result [31..0]</li> <li>• clk</li> <li>• clk_en</li> <li>• start</li> <li>• reset</li> <li>• done</li> <li>• n [7..0]</li> </ul>

**Table 1–1. Custom Instruction Architectural Types, Application and Hardware Ports**

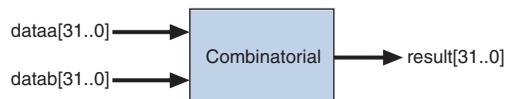
Architectural Type	Application	Hardware Ports
Internal Register File	Custom logic blocks that access internal register files for input and/or output	<ul style="list-style-type: none"> <li>• dataa[31..0]</li> <li>• datab[31..0]</li> <li>• result[31..0]</li> <li>• clk</li> <li>• clk_en</li> <li>• start</li> <li>• reset</li> <li>• done</li> <li>• n[7..0]</li> <li>• a[4..0]</li> <li>• readra</li> <li>• b[4..0]</li> <li>• readrb</li> <li>• c[4..0]</li> <li>• writerc</li> </ul>
External Interface	Custom logic blocks that interface to logic outside of the Nios II processor's data path	Standard custom instruction ports, plus user-defined interface to external logic.

This section discusses the basic functionality and hardware interface of each custom instruction architecture type listed in [Table 1–1](#).

### Combinatorial Custom Instruction Architecture

Combinatorial custom instruction architecture consists of a logic block that is able to complete in a single clock cycle.

[Figure 1–3](#) shows a block diagram of a combinatorial custom instruction architecture.

**Figure 1–3. Combinatorial Custom Instruction Architecture**

The [Figure 1–3](#) combinatorial custom instruction diagram uses the `dataa` and `datab` ports as inputs and drives the results on the `result` port. Because the logic is able to complete in a single clock cycle, control ports are not needed.

Table 1–2 lists the combinatorial custom instruction ports.

<i>Table 1–2. Combinatorial Custom Instruction Ports</i>			
<b>Port Name</b>	<b>Direction</b>	<b>Required</b>	<b>Purpose</b>
dataa [31..0]	Input	No	Input operand to custom instruction
datab [31..0]	Input	No	Input operand to custom instruction
result [31..0]	Output	Yes	Result from custom instruction

The only required port for combinatorial custom instructions is the `result` port. The `dataa` and `datab` ports are optional. Include them only if the custom instruction functionality requires input operands. If the custom instruction requires only a single input port, use `dataa`.

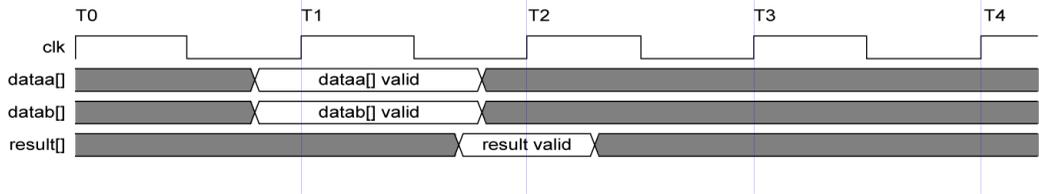
### *Combinatorial Port Operation*

This section describes the combinatorial custom instruction hardware port operation. Figure 1–4 shows the combinatorial custom instruction hardware port timing diagram.

In Figure 1–4, the processor presents the input data on the `dataa` and `datab` ports on the rising edge of the processor clock. The processor reads the `result` port on the rising edge of the following processor clock.

The Nios II processor issues a combinatorial custom instruction speculatively; that is, it optimizes execution by issuing the instruction before knowing whether it is necessary, and ignores the result if it is not required. Therefore, a combinatorial custom instruction must not have side effects. In particular, a combinatorial custom instruction cannot have an external interface.

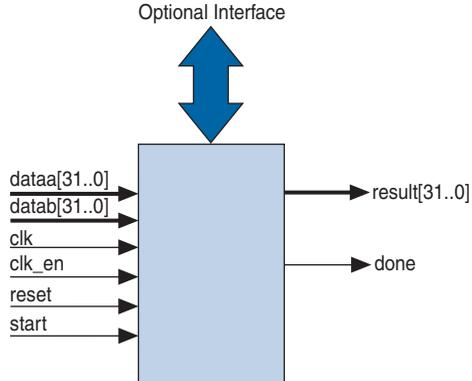
You can further optimize combinatorial custom instructions by implementing the extended custom instruction architecture. Refer to “Extended Custom Instruction Architecture” on page 1–9.

**Figure 1–4. Combinatorial Custom Instruction Port Timing Diagram**

## Multi-Cycle Custom Instruction Architecture

Multi-cycle, or sequential, custom instructions consist of a logic block that requires two or more clock cycles to complete an operation. Additional control ports are required for multi-cycle custom instructions. See [Table 1–3](#).

[Figure 1–5](#) shows the multi-cycle custom instruction block diagram.

**Figure 1–5. Multi-Cycle Custom Instruction Block Diagram**

Multi-cycle custom instruction can complete in either a fixed or variable number of clock cycles.

- **Fixed length:** You specify the required number of clock cycles during system generation
- **Variable length:** The `start` and `done` ports are used in a handshaking scheme to determine when the custom instruction execution is complete.

Table 1–3 lists multi-cycle custom instruction ports.

Port Name	Direction	Required	Application
clk	Input	Yes	System clock
clk_en	Input	Yes	Clock enable
reset	Input	Yes	Synchronous reset
start	Input	No	Commands custom instruction logic to start execution
done	Output	No	Custom instruction logic indicates to the processor that execution is complete.
dataa[31..0]	Input	No	Input operand to custom instruction
datab[31..0]	Input	No	Input operand to custom instruction
result[31..0]	Output	No	Result from custom instruction

As indicated in Table 1–3, the `clk`, `clk_en`, and `reset` ports are required for multi-cycle custom instructions. However, the `start`, `done`, `dataa`, `datab`, and `result` ports are optional. Implement them only if the custom instruction functionality specifically needs them.

### *Multi-Cycle Port Operation*

The section provides operational details for the multi-cycle custom instruction hardware port. Figure 1–6 shows the multi-cycle custom instruction timing diagram.

- The processor asserts the active high `start` port on the first clock cycle of execution when the custom instruction issues through the ALU. At this time, the `dataa` and `datab` ports have valid values and remain valid throughout the duration of the custom instruction execution.
- Fixed or variable length custom instruction port operation:
  - Fixed length: The processor asserts `start`, waits a specified number of clock cycles, and then reads `result`. For an  $n$ -cycle operation, the custom logic block must present valid data on the  $(n-1)^{\text{st}}$  rising edge after the `start` port is asserted.
  - Variable length: The processor waits until the active high `done` port is asserted. The processor reads the `result` port on the clock edge that `done` is asserted. The custom logic block must present data on the `result` port on the same clock cycle that the `done` port is asserted.

- The Nios II system clock feeds the custom logic block's `clk` port, and the Nios II system's master reset feeds the active high `reset` port. The `reset` port is asserted only when the whole Nios II system is reset.
- The custom logic block must treat the active high `clk_en` port as a conventional clock qualifier signal, ignoring `clk` while `clk_en` is deasserted.
- Any port in the custom logic block that is not recognized as a custom instruction port is considered to be an external interface.
- You can further optimize multi-cycle custom instructions by implementing the extended internal register file, or by creating external interface custom instructions. Refer to “[Extended Custom Instruction Architecture](#)” on page 1-9, “[Internal Register File Custom Instruction Architecture](#)” on page 1-11, or “[External Interface Custom Instruction Architecture](#)” on page 1-12.

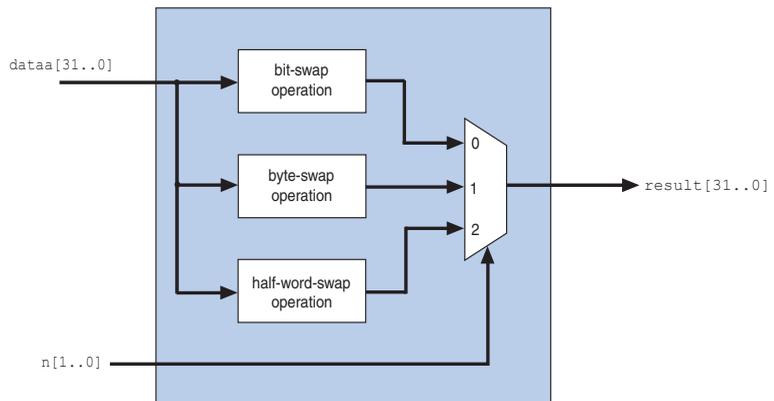
**Figure 1-6. Multi-Cycle Custom Instruction Timing Diagram**



## Extended Custom Instruction Architecture

Extended custom instruction architecture allows a single custom logic block to implement several different operations. Extended custom instructions use the `N` field to specify which operation the logic block performs. The `N` field of the instruction word can be up to eight bits wide, allowing a single custom logic block to implement up to 256 different operations.

[Figure 1-7](#) is a block diagram of an extended custom instruction with bit swap, byte swap, and half-word swap operations.

**Figure 1–7. Extended Custom Instruction with Swap Operations**

The custom instruction in [Figure 1–7 on page 1–10](#) performs swap operations on data received at the `dataa` port. It uses the two-bit-wide `n` port to select the output from a multiplexer, determining which result is presented to the `result` port. Input to the `n` port comes directly from the `N` field of the custom instruction word.



This logic is just a simple example, using a multiplexer on the output. You can implement function selection based on the `N` field in any way that is appropriate for your application.

Extended custom instructions can be combinatorial or multi-cycle custom instructions. To implement an extended custom instruction, simply add an `n` port to your custom instruction logic. The bit width of the `n` port is a function of the number of operations the custom logic block can perform.

Extended custom instructions occupy multiple custom instruction indices. For example, the custom instruction illustrated in [Figure 1–7 on page 1–10](#) occupies 4 indices, because `n` is two bits wide. Therefore, when this instruction is implemented in a Nios II system, it leaves  $256 - 4 = 252$  available indices. For information about the custom instruction index, see [“Custom Instruction Assembly Software Interface” on page 2–4](#).

### *Extended Custom Instruction Port Operation*

The `n` port behaves similarly to the `dataa` port. The processor presents the `n` port on the rising edge of the clock when `start` is asserted, and the `n` port remains stable throughout the execution of the custom instruction.

All other custom instruction port operations remain the same.

## Internal Register File Custom Instruction Architecture

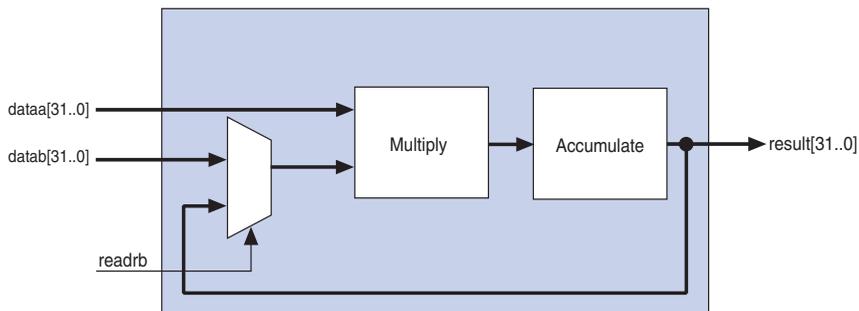
The Nios II processor allows custom instruction logic to access its own internal register file for I/O, which provides you the flexibility to specify if the custom instruction reads its operands from the Nios II processor's register file or from the custom instruction's own internal register file. In addition, a custom instruction can write its results to the local register file rather than the Nios II processor's register file.

Internal registers accessing custom instructions use `readra`, `readrb`, and `writerc` to determine if I/O takes place between the Nios II processor's register file or an internal register file. Additionally, ports `a`, `b`, and `c` specify which internal registers to read from and/or write to. For example, if `readra` is deasserted (that is, read from the internal register), `a` provides an index to the internal register file.

For further details of Nios II custom instruction implementation, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

Figure 1–8 shows a simple multiply-accumulate custom logic block.

**Figure 1–8. Multiply-Accumulate Custom Logic Block**



When `readrb` is deasserted, the logic block multiplies `dataa` and `datab`, and stores the results in the accumulate register. The Nios II processor can read back those results. Alternatively, the processor can read the value in the accumulator as input to the multiplier by asserting `readrb`.

Table 1–4 lists the internal register file custom instruction ports. Use these optional ports only if the custom instruction functionality requires them.

**Table 1–4. Internal Register File Custom Instruction Ports**

Port Name	Direction	Required	Application
readra	Input	No	If readra is high, the Nios II processor supplies dataa and datab. If readra is low, custom instruction logic reads the internal register file indexed by a.
readrb	Input	No	If readrb is high, the Nios II processor supplies dataa and datab. If readrb is low, custom instruction logic reads the internal register file indexed by a.
writerc	Input	No	Causes custom instruction to write result of c to custom instruction internal register file.
a	Input	No	Custom instruction internal register file index
b	Input	No	Custom instruction internal register file index
c	Input	No	Custom instruction internal register file index

### Internal Register File Custom Instruction Port Operation

The readra, readrb, writerc, and a, b, and c ports behave similarly to dataa. When the start port is asserted, the processor presents the readra, readrb, writerc, a, b, and c ports on the rising edge of the processor clock. All the ports remain stable throughout the execution of the custom instructions.

To determine how to handle register file I/O, custom instruction logic reads the active high readra, readrb, and writerc ports. The logic uses the a, b, and c ports as register file indexes. When readra or readrb are not asserted, the custom instruction logic ignores the corresponding a or b port. When writerc is not asserted, the processor ignores the value driven on the result port.

All other custom instructions port operations remain the same.

### External Interface Custom Instruction Architecture

Figure 1–9 shows that the Nios II custom instructions allow you to add an interface to communicate with logic outside of the processor’s data path. At system generation, any interfaces that are not recognized as custom instruction ports propagate out to the top level of the SOPC Builder module where external logic can access the interfaces.

Because the custom instruction logic is able to access memory external to the processor, it extends the capabilities of the custom instruction logic.

**Figure 1–9. Custom Instructions Allow the Addition of an External Interface**

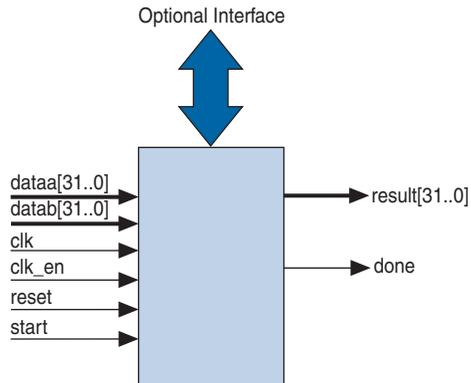


Figure 1–9 shows a multi-cycle custom instruction that has an external memory interface.

Custom instruction logic can perform various tasks, for example, store intermediate results, or read memory to control the custom instruction operation. The optional external interface also provides a dedicated path for data to flow into, or out of, the processor. For example, custom instruction logic can feed data directly from the processor's register file to an external first-in first-out (FIFO) memory buffer, bypassing the processor's data bus.



### Introduction

The Nios II custom instruction software interface abstracts logic implementation details from the application code. During the build process the Nios II IDE generates macros that allow easy access from application code to custom instructions.

This chapter provides custom instruction software interface details including:

- “Custom Instruction Examples” on page 2-1
- “Built-In Functions and User-Defined Macros” on page 2-2
- “Custom Instruction Assembly Software Interface” on page 2-4

### Custom Instruction Examples

Example 2-1 shows a portion of the **system.h** header file that defines the macro for a bit swap custom instruction. This bit swap example uses one 32-bit input and performs only one function.

---

#### Example 2-1. Bit Swap Macro Definition

```
#define ALT_CI_BSWAP_N 0x00  
  
#define ALT_CI_BSWAP(A) __builtin_custom_ini(ALT_CI_BSWAP_N, (A))
```

---

In Example 2-1, `ALT_CI_BSWAP_N` is defined to be `0x0`, which is the custom instruction’s index. The `ALT_CI_BSWAP(A)` macro is mapped to a gcc built-in function that takes a single argument.

Example 2-2 illustrates a bit swap custom instruction used in application code.

### Example 2–2. Bit Swap Instruction Usage

```
1. #include "system.h"
2.
3.
4. int main (void)
5. {
6.     int a = 0x12345678;
7.     int a_swap = 0;
8.
9.     a_swap = ALT_CI_BSWAP(a);
10.    return 0;
11.}
```

---

In Example 2–2, the **system.h** file is included to define the custom instruction macro definitions. The example declares two integers, `a` and `a_swap`. Integer `a` is passed as input to the bit swap custom instruction with the results loaded into `a_swap`.

Example 2–2 accommodates most applications using custom instructions. The macros defined by the Nios II IDE only make use of C integer types. Occasionally, applications need to make use of input types other than integers, and therefore, need to pass expected return values other than integers.



You can define custom macros for Nios II custom instructions, that allow for other 32-bit input types to interface with custom instructions.

## Built-In Functions and User-Defined Macros

The Nios II processor uses `gcc` built-in functions to map to custom instructions. By using built-in functions, software can use non-integer types with custom instructions. There are 52 uniquely-defined built-in functions to accommodate the different combinations of the supported types.

Built-in function names have the following format:

`__builtin_custom_<return type>n<parameter types>`

Table 2-1 shows 32-bit types supported by custom instructions as parameters and return types, as well as the abbreviations used in the built-in function name.

Type	Built-In Function Abbreviation
int	i
float	f
void *	p

Example 2-3 shows the prototype definitions for two built-in functions.

---

### **Example 2-3. Built-in Functions**

```
void __builtin_custom_nf (int n, float dataa);
float __builtin_custom_fnp (int n, void * dataa);
```

---

In Example 2-3, the `__builtin_custom_nf` function takes an `int` and a `float` as inputs, and does not return a value. In contrast, the `__builtin_custom_fnp` function takes a pointer as an input, and returns a `float`.

To support non-integer input types, define macros with mnemonic names that map to the specific built-in function required for the application.



Refer to [Appendix B, Custom Instruction Built-In Functions](#) for detailed information, and a list of built-in functions.

Example 2-4 shows user-defined custom instruction macros used in an application.

### Example 2–4. Custom Instruction Macro Usage

```
1. /* define void udef_macro1(float data); */
2. #define UDEF_MACRO1_N 0x00
3. #define UDEF_MACRO1(A) __builtin_custom_nf(UDEF_MACRO1_N, (A));
4. /* define float udef_macro2(void *data); */
5. #define UDEF_MACRO2_N 0x01
6. #define UDEF_MACRO2(B) __builtin_custom_fnp(UDEF_MACRO2_N, (B));
7.
8. int main (void)
9. {
10. float a = 1.789;
11. float b = 0.0;
12. float *pt_a = &a;
13.
14. UDEF_MACRO1(a);
15. b = UDEF_MACRO2((void *)pt_a);
16. return 0;
17. }
```

---

On lines 2 through 6, the user-defined macros are declared and mapped to the appropriate built-in functions. The macro `UDEF_MACRO1` takes a float as an input parameter and does not return anything. The macro `UDEF_MACRO2` takes a pointer as an input parameter and returns a float. Lines 14 and 15 show the use of the two user-defined macros.

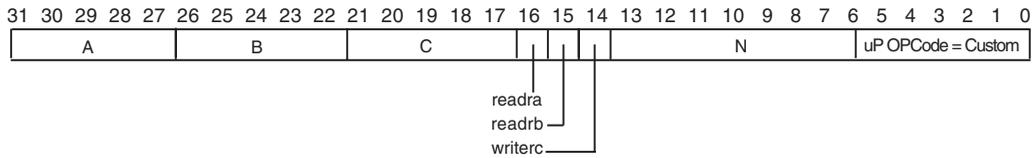
## Custom Instruction Assembly Software Interface

The Nios II custom instructions are also accessible in assembly code. This section describes the assembly interface.

Custom instructions are R-type instructions, containing:

- A 6-bit opcode
- Three 5-bit register index fields
- Three 1-bit fields for the `readra`, `readrb` and `writerc` ports
- An 8-bit N field, used for the custom instruction index (opcode extension), and optionally including a function select subfield

Figure 2–1 on page 2–5 is a diagram of the custom instruction word.

**Figure 2–1. Custom Instruction Word**

**Instruction Fields:** A = Register index of operand A  
 B = Register index of operand B  
 C = Register index of operand C  
 N = 8-bit number that selects instruction  
 readra = 1 if instruction uses rA, 0 otherwise  
 readrb = 1 if instruction uses rB, 0 otherwise  
 writerc = 1 if instruction provides result for rC, 0 otherwise

Bits 5–0 are the Nios II custom instruction opcode, as specified in the “*Instruction Opcodes*” section in the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*. This value appears in every custom instruction.

The N field, bits 13–6, is the custom instruction index. The custom instruction index distinguishes between different custom instructions, allowing the Nios II processor to support up to 256 distinct custom instructions. Depending on the type of custom instruction, the N field represents one of the following:

- A unique custom instruction index, for logic that implements a single custom function
- An extended custom instruction index, for logic that implements several custom functions

Example 2–5 shows the assembly language syntax for the custom instruction.

---

### Example 2–5. Custom Instruction Syntax

```
custom N, xC, xA, xB
```

In Example 2–5, N is the custom instruction index, xC is the destination for the result [31..0] port, xA is the dataa port, and xB is the datab port. To access the Nios II processor’s register file, replace x with r. To access a custom register file, replace x with c.

Examples 2-6 and 2-7 show the syntax for two examples of custom instruction assembler calls.

---

### **Example 2-6. Custom Instruction Index=0**

```
custom 0, r6, r7, r8
```

---

Example 2-6 executes a custom instruction with an index of 0. The contents of the Nios II processor registers r7 and r8 are used as input, with the results stored in the Nios II processor register r6.

---

### **Example 2-7. Custom Instruction Index=3**

```
custom 3, c1, r2, c4
```

---

Example 2-7 executes a custom instruction with an index of 3. The contents of the Nios II processor register r2 and custom register c4 are used as inputs. The results are stored in the custom register c1.



For further information about the binary format of custom instructions, refer to the *Instruction Set Reference* chapter of the *Nios II Processor Reference Handbook*.

### Introduction

This chapter describes the process of implementing a Nios II custom instruction with the SOPC Builder component editor. The component editor enables you to create new SOPC Builder components, including Nios II custom instructions.



For detailed information about the SOPC Builder component editor, refer to the *Component Editor* chapter of the *Quartus II Handbook Volume 4: SOPC Builder*.

### Implementing Custom Instruction Hardware in SOPC Builder

Implementing a Nios II custom instruction entails the following tasks:

- “Open the Component Editor” on page 3-1
- “Add the Top-Level HDL File” on page 3-3
- “Import Simulation Files” on page 3-4
- “Configure the Custom Instruction Ports” on page 3-4
- “Set Up Additional Interfaces” on page 3-5
- “Set the Component Group Name” on page 3-6
- “Save the Custom Instruction” on page 3-7
- “Generate the SOPC Builder System and Compile in Quartus II Software” on page 3-9

The following sections detail the steps required to carry out these tasks. This process imports the custom instruction into the design, and adds it to the Nios II processor.

#### Open the Component Editor

1. Open the SOPC Builder system.

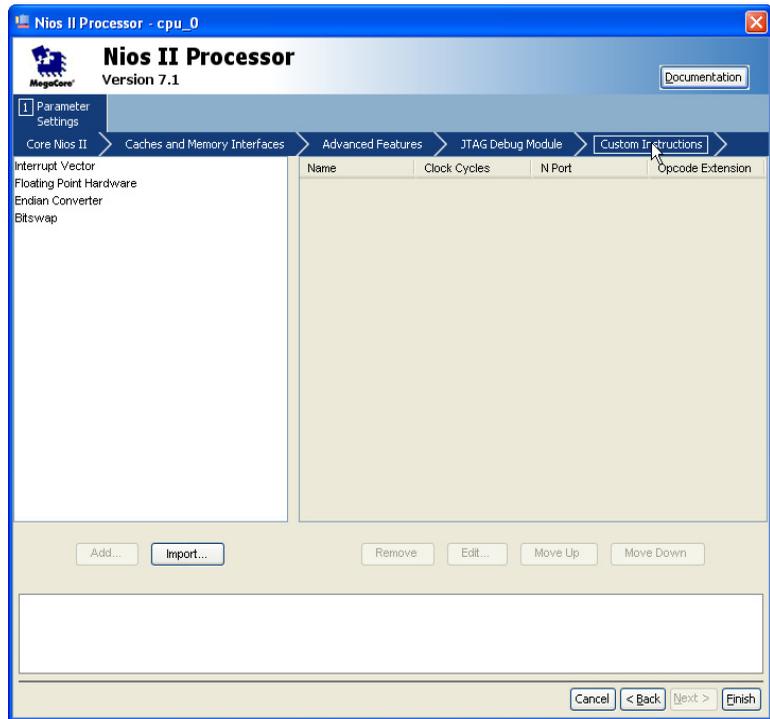


For detailed information about opening and working with SOPC Builder systems, refer to the *Quartus II Handbook Volume 4: SOPC Builder*, or to the SOPC Builder Help system.

2. Select the Nios II processor in the **Altera SOPC Builder System Contents** page.
3. In the Module menu, click **Edit....** The Nios II configuration wizard appears.

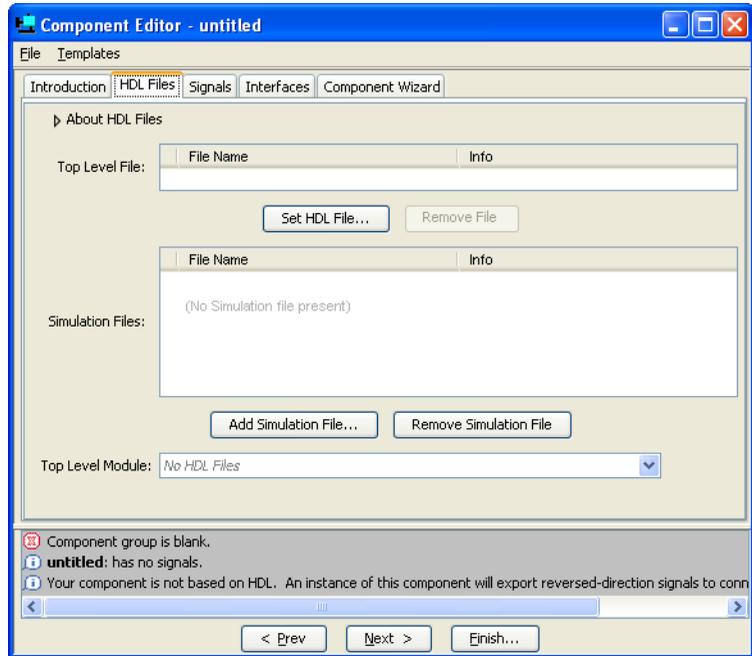
4. Select the **Custom Instructions** page, shown in [Figure 3–1](#)

**Figure 3–1. Nios II Configuration Wizard Custom Instructions Page**



5. Click **Import...** The SOPC Builder component editor appears, displaying the Introduction tab.

Figure 3–2. Component Editor



### Add the Top-Level HDL File

1. Click **Next** to display the HDL Files tab, shown in [Figure 3–2](#).
2. Click **Set HDL File**.
3. Browse to the directory containing the top-level hardware definition language (HDL) file, and select it.
4. Click **Open**. The component editor shows the HDL file in the Top Level File list.

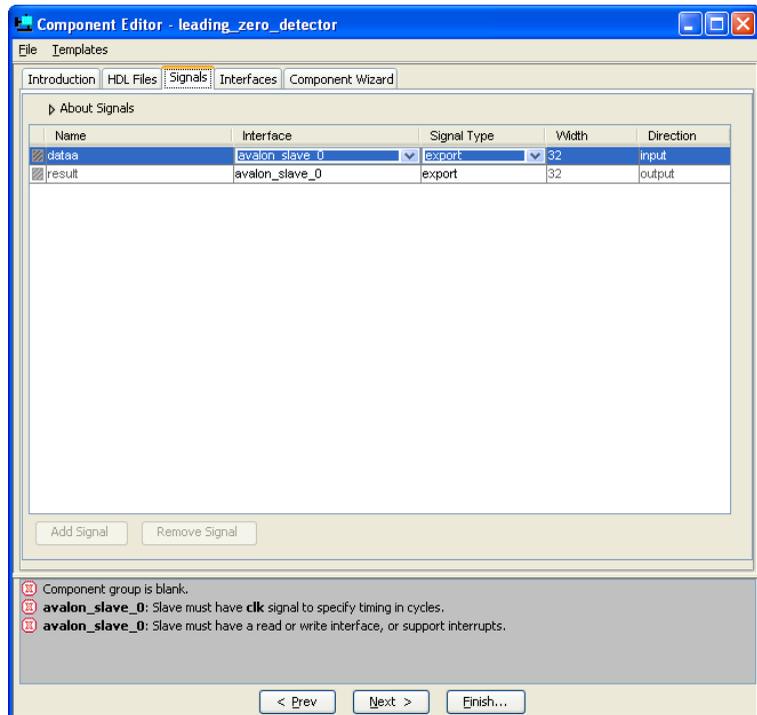
The Quartus II Analyzer checks the design for errors in the background. The HDL file list blinks while analysis is taking place. When analysis is complete, a dialog box reports **Info: Quartus II Analysis & Synthesis was successful**.

5. Click **OK**.

## Import Simulation Files

If you are running the system in ModelSim, and you have additional simulation files, import them by using the **Add Simulation File...** button.

**Figure 3–3. Component Editor: Signals Tab**



## Configure the Custom Instruction Ports

1. Click **Next** to display the Signals tab, shown in [Figure 3–3](#). There are several ports (signals) listed.
2. For the first port, carry out the following steps:
  - a. Select the port.
  - b. In the drop-down box under Interface, select **New nios\_custom\_instruction\_slave**. The component editor creates a new interface, **nios\_custom\_instruction\_slave\_<n>**, which appears in the drop-down box.

The component editor names each Nios II custom instruction sequentially, starting with  $\langle n \rangle = 0$ .

- c. In the drop-down box under Signal Type, select the signal type corresponding to the port name. For example, if the custom instruction hardware presents the result on a port named `output`, you set the type of `output` to `result`.

For further information about signal types, see [“Custom Instruction Architectural Types” on page 1–4](#).

3. For each additional port, carry out the following steps:
  - a. Select the port.
  - b. In the drop-down box under Interface, select `nios_custom_instruction_slave_<n>`, the Nios II custom instruction slave interface you created in step 2.
  - c. In the drop-down box under Signal Type, select the signal type corresponding to the port name.

## Set Up Additional Interfaces

Exported signal types are considered to be a part of the custom instruction’s external interface. This section describes how to set up custom interfaces.

For further information about external interface custom instructions, see [“Custom Instruction Architectural Types” on page 1–4](#).

If your custom instruction hardware requires additional interfaces, either to the Avalon-MM system interconnect fabric or outside the SOPC Builder system, you can specify these interfaces here.

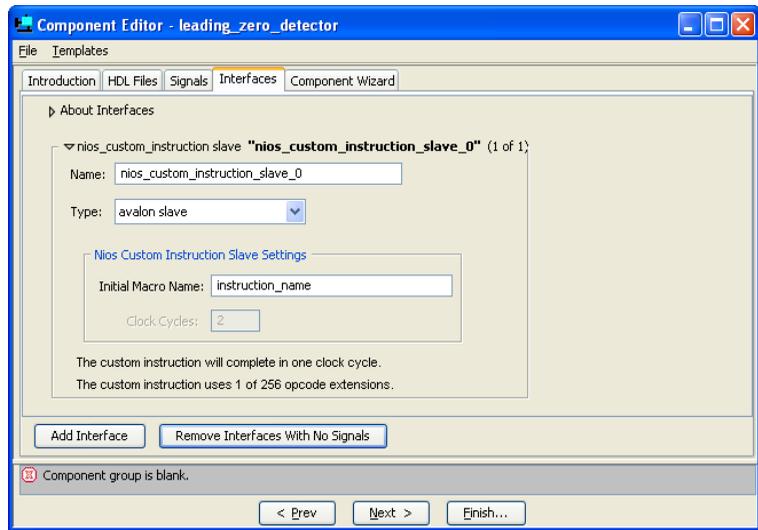


Most custom instructions use some combination of standard custom instruction ports, such as `dataa`, `datab`, and `result`, and do not require any additional interfaces.

Also custom instructions can be published for later reuse in different projects.

1. Click **Next** to display the Interfaces tab, shown in [Figure 3–4 on page 3–6](#).

**Figure 3–4. Component Editor: Interfaces Tab**

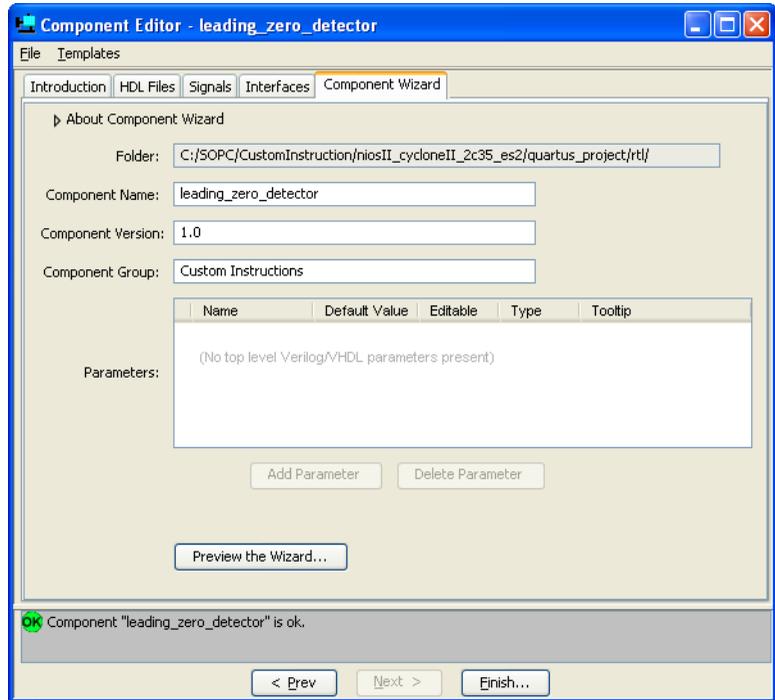


2. If there is a message reporting **Interface has no signals**, click **Remove Interfaces With No Signals**. The message disappears.

### Set the Component Group Name

1. Click **Next** to display the Component Wizard tab.
2. In the Component Group text box, type Custom Instructions, as shown in [Figure 3–5](#).

Figure 3–5. Component Editor: Component Wizard Tab



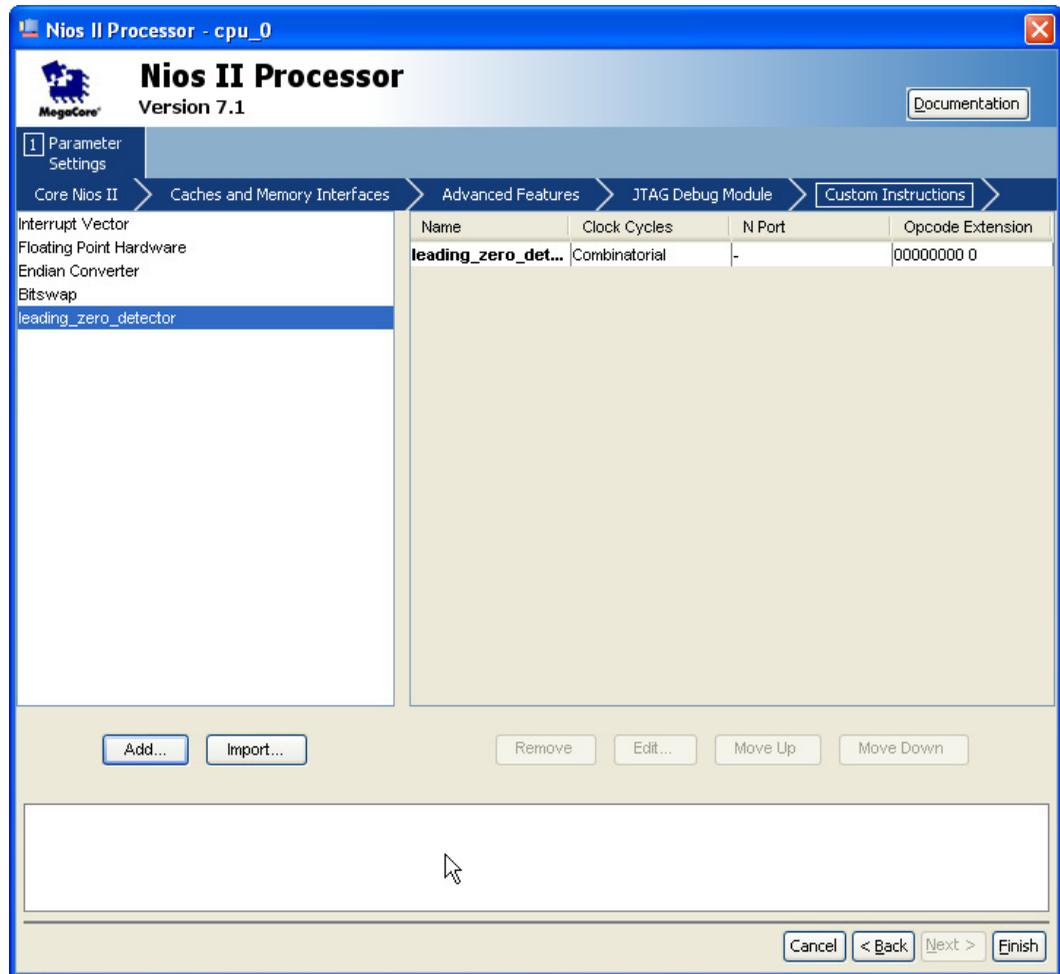
3. Make sure the bottom pane of the dialog box displays the message **Component "<design name>" is ok**. If it does not, review the preceding steps.

### Save the Custom Instruction

1. Click **Finish**. A dialog box states **You are about to save <design name> 1.0 to: ...** followed by the name of the directory containing the top-level HDL file.
2. Click **Yes** to finish importing the custom instruction and return to the Nios II configuration wizard.
3. Select the new custom instruction, and click **Add** to add it to the Nios II processor.

As shown in Figure 3–6, after you add the custom instruction to the processor, the **Name** field lists the top level module name.

Figure 3–6. Altera Nios II Configuration Wizard with Custom Instruction



The **Clock Cycles** field indicates the type of custom instruction: combinatorial logic, multi-cycle, extended, internal register file, or external interface. If the custom instruction is a fixed length, multi-cycle custom instruction, you must edit this field to specify the number of clocks. You must determine this number based on knowledge of the custom instruction state machine. In the case of a variable length multi-cycle custom instruction, the **Clock Cycles** field displays **Variable**.

The **N port** field indicates whether the custom instruction is an extended type. In the case of an extended custom instruction, this field indicates which bits in the n port serve as the function select. Otherwise it displays a dash (-).

The **Opcode Extension** field displays the custom instruction index (N field) in the instruction word. The value appears in both binary and decimal formats. For further information about the N field, see [“Custom Instruction Assembly Software Interface”](#) on page 2–4.

4. In the Nios II configuration wizard, click **Finish** to finish adding the custom instruction to the system and return to the SOPC Builder window.

## Generate the SOPC Builder System and Compile in Quartus II Software

After the custom instruction logic is added to the system, you are ready for system generation and Quartus II compilation. During system generation, SOPC Builder connects the custom logic to the Nios II processor.



For detailed instructions on generating SOPC Builder systems, refer to the *Quartus II Handbook Volume 4: SOPC Builder*, or to the SOPC Builder Help system.

## Accessing the Custom Instruction from Software

Adding a custom instruction to a Nios II processor results in a significant change to the SOPC Builder system. The next time you run any related Nios II program in the Nios II IDE, the IDE rebuilds the software project to accommodate the changes. The IDE adds the macros for the custom instruction to the `system.h` header file.

For details about writing software for Nios II custom instructions, see [Chapter 2, Software Interface](#).





## Overview

This section provides VHDL and Verilog HDL custom instruction wrapper file templates that you can reference when writing custom instructions in VHDL and Verilog HDL.



You can download the template files from the Altera web site at [www.altera.com/nios](http://www.altera.com/nios).

## VHDL Template

Sample VHDL template file:

```
LIBRARY <__library_name>;
USE <__library_name>.<__package_name>.ALL;

ENTITY <__entity_name> IS
  PORT(
    signal clk : IN STD_LOGIC; -- CPU's master-input clk <required for multi-cycle>
    signal reset : IN STD_LOGIC; -- CPU's master asynchronous reset <required for multi-cycle>
    signal clk_en: IN STD_LOGIC; -- Clock-qualifier <required for multi-cycle>
    signal start: IN STD_LOGIC; -- True when this instr. issues <required for multi-cycle>
    signal done; OUT STD_LOGIC; -- True when instr. completes <required for variable multi-cycle>
    signal dataa: IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand A <always required>
    signal datab: IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- operand B <optional>
    signal n: IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- N-field selector <required for extended>
    signal a: IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- operand A selector <used for Internal register
      file access>
    signal b: IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- operand B selector <used for Internal register
      file access>
    signal c: IN STD_LOGIC; -- result destination selector <used for Internal register file
      access>
    signal readra: IN STD_LOGIC; -- register file index <used for Internal register file access>
    signal readrb: IN STD_LOGIC; -- register file index <used for Internal register file access>
    signal writerc: IN STD_LOGIC; -- register file index <used for Internal register file access>
    signal result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) -- result <always required>
  );
END <__entity_name>;

ARCHITECTURE a OF <__entity_name> IS
  signal clk: IN STD_LOGIC;
  signal reset : IN STD_LOGIC;
  signal clk_en: IN STD_LOGIC;
  signal start: IN STD_LOGIC;
  signal readra: IN STD_LOGIC;
  signal readrb: IN STD_LOGIC;
  signal writerc: IN STD_LOGIC;
  signal n: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
  signal a: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
  signal b: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
  signal c: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
  signal dataa: IN STD_LOGIC_VECTOR (31 DOWNTO 0);
  signal datab: IN STD_LOGIC_VECTOR (31 DOWNTO 0);

  signal result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
```

```
    signal done: OUT STD_LOGIC;
BEGIN

-- Process Statement
-- Concurrent Procedure Call
-- Concurrent Signal Assignment
-- Conditional Signal Assignment
-- Selected Signal Assignment
-- Component Instantiation Statement
-- Generate Statement

END a;
```

## Verilog HDL Template

Sample Verilog HDL template file:

```
//Verilog Custom Instruction Template

module <_module_name>(
    clk, // CPU's master-input clk <required for multi-cycle>
    reset, // CPU's master asynchronous reset <required for multi-cycle>
    clk_en, // Clock-qualifier <required for multi-cycle>
    start, // True when this instr. issues <required for multi-cycle>
    done, // True when instr. completes <required for variable multi-cycle>
    dataa, // operand A <always required>
    datab, // operand B <optional>
    n, // N-field selector <required for extended>
    a, // operand A selector <used for Internal register file access>
    b, // operand b selector <used for Internal register file access>
    c, // result destination selector <used for Internal register file access>
    readra, // register file index <used for Internal register file access>
    readrb, // register file index <used for Internal register file access>
    writerc, // register file index <used for Internal register file access>
    result // result <always required>
);

input clk;
input reset;
input clk_en;
input start;
input readra;
input readrb;
input writerc;
input [7:0] n;
input [4:0] a;
input [4:0] b;
input [4:0] c;
input [31:0] dataa;
input [31:0] datab;

output [31:0] result;
output done;

// Port Declaration

// Wire Declaration

// Integer Declaration

// Concurrent Assignment
```

---

```
// Always Construct  
endmodule
```



## Overview

The Nios II gcc compiler is customized with built-in functions to support custom instructions. This section lists the built-in functions.



For more information about gcc built-in functions, refer to [www.gnu.org](http://www.gnu.org).

Nios II custom instruction built-in functions are of the following types:

- Returning void
- Returning int
- Returning float
- Returning a pointer

## Built-In Functions Returning void

```
void __builtin_custom_n (int n);
void __builtin_custom_ni (int n, int dataa);
void __builtin_custom_nf (int n, float dataa);
void __builtin_custom_np (int n, void *dataa);
void __builtin_custom_nii (int n, int dataa, int datab);
void __builtin_custom_nif (int n, int dataa, float datab);
void __builtin_custom_nip (int n, int dataa, void *datab);
void __builtin_custom_nfi (int n, float dataa, int datab);
void __builtin_custom_nff (int n, float dataa, float datab);
void __builtin_custom_nfp (int n, float dataa, void *datab);
void __builtin_custom_npi (int n, void *dataa, int datab);
void __builtin_custom_npf (int n, void *dataa, float datab);
void __builtin_custom_npp (int n, void *dataa, void *datab);
```

## Built-In Functions Returning int

```
int __builtin_custom_in (int n);
int __builtin_custom_ini (int n, int dataa);
int __builtin_custom_inf (int n, float dataa);
int __builtin_custom_inp (int n, void *dataa);
int __builtin_custom_inii (int n, int dataa, int datab);
int __builtin_custom_inif (int n, int dataa, float datab);
int __builtin_custom_inip (int n, int dataa, void *datab);
int __builtin_custom_infi (int n, float dataa, int datab);
int __builtin_custom_inff (int n, float dataa, float datab);
int __builtin_custom_infp (int n, float dataa, void *datab);
int __builtin_custom_inpi (int n, void *dataa, int datab);
int __builtin_custom_inpf (int n, void *dataa, float datab);
int __builtin_custom_inpp (int n, void *dataa, void *datab);
```

## Built-in Functions Returning float

```
float __builtin_custom_fn (int n);
float __builtin_custom_fni (int n, int dataa);
float __builtin_custom_fnf (int n, float dataa);
float __builtin_custom_fnp (int n, void *dataa);
float __builtin_custom_fnii (int n, int dataa, int datab);
float __builtin_custom_fnif (int n, int dataa, float datab);
float __builtin_custom_fnip (int n, int dataa, void *datab);
float __builtin_custom_fnfi (int n, float dataa, int datab);
float __builtin_custom_fnff (int n, float dataa, float datab);
float __builtin_custom_fnfp (int n, float dataa, void *datab);
float __builtin_custom_fnpi (int n, void *dataa, int datab);
float __builtin_custom_fnpf (int n, void *dataa, float datab);
float __builtin_custom_fnpp (int n, void *dataa, void *datab);
```

## Built-in Functions Returning a Pointer

```
void *__builtin_custom_pn (int n);
void *__builtin_custom_pni (int n, int dataa);
void *__builtin_custom_pnf (int n, float dataa);
void *__builtin_custom_pnp (int n, void *dataa);
void *__builtin_custom_pnii (int n, int dataa, int datab);
void *__builtin_custom_pnif (int n, int dataa, float datab);
void *__builtin_custom_pnip (int n, int dataa, void *datab);
void *__builtin_custom_pnfi (int n, float dataa, int datab);
void *__builtin_custom_pnff (int n, float dataa, float datab);
void *__builtin_custom_pnfp (int n, float dataa, void *datab);
void *__builtin_custom_pnpi (int n, void *dataa, int datab);
void *__builtin_custom_pnpf (int n, void *dataa, float datab);
void *__builtin_custom_pnpp (int n, void *dataa, void *datab);
```



# Appendix C. Porting First-Generation Nios Custom Instructions to Nios II Systems

## Overview

Most first-generation Nios custom instructions will port over to a Nios II system with minimal changes. This chapter clarifies hardware and software considerations when porting first-generation Nios custom instructions to your Nios II system.

## Hardware Porting Considerations

Both combinatorial and multi-cycle first-generation Nios custom instructions will work with a Nios II system without any changes. However, because parameterized first-generation Nios custom instructions allow a prefix to be passed to the custom instruction logic block, parameterized first-generation Nios custom instructions require a design change.

There is no strict definition for the use of prefixes in first-generation Nios systems, but in most cases the prefix controls the operation performed by the custom instruction. However in a Nios II system, the prefix option is supported directly by extended custom instructions. Therefore, any parameterized first-generation Nios custom instruction that uses a prefix to control the operation executed by the custom instruction should be ported to a Nios II extended custom instruction. Refer to [“Extended Custom Instruction Architecture” on page 1–9](#).

Any other use of the prefix can be accomplished with one of the Nios II custom instruction architecture types. Refer to [“Custom Instruction Architectural Types” on page 1–4](#).

## Software Porting Considerations

All first-generation Nios custom instructions will require a small change to application software. Assuming no hardware changes (i.e., not a parameterized first-generation custom instruction), software porting should be nothing more than a search and replace operation. The first-generation Nios and Nios II system macro definition nomenclature is different; therefore first-generation Nios macro calls should be replaced by the Nios II macros. In the case of parameterized first-generation custom instructions, additional changes will be required depending on the implementation. Refer to [Chapter 2, Software Interface](#).

